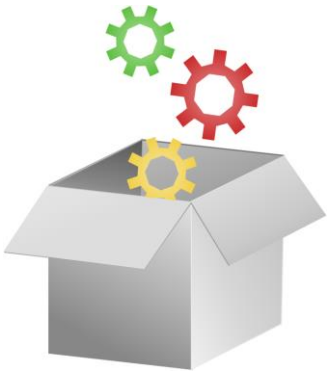
Method SCRIPT SDK Example



PalmSens
Compact Electrochemical Interfaces

Last revision: March ? 2019

Commented [P1]: Tbu

www.palmsens.com

1.1 Contents:

The arduino example “MethodScriptExample.ino” found in the “/MethodScriptExample-Arduino” folder demonstrates basic communication with the EmStat Pico through Arduino MKR ZERO using the embedded SDK (C libraries). The example allows the user to start measurements on the EmStat Pico from the PC connected to the Arduino through USB.

1.2 Hardware setup:

- To run this example, connect your Arduino MKRZERO "Serial1" port Rx (pin 13), Tx (pin 14) and GND to the EmStat Pico “Serial” Tx, Rx and GND respectively.
- Make sure the UART switch block SW4 on the EmStat dev board has the switches for MKR 3 and 4 turned on.
- The Arduino board should be connected normally to a PC.
- If not powering the EmStat by other means, the EmStat Pico should be connected to the PC through USB for power.

1.3 Environment setup:

- To run this example, you must include the Method SCRIPT C libraries first.
- To do this, follow the menu "Sketch -> Include Library -> Add .ZIP/Library..." and select the MethodScriptComm folder.

1.4 How to use

- Compile and upload this sketch through the Arduino IDE.
- Next, open a serial monitor to the Arduino (you can do this from the Arduino IDE).
- You should see messages being printed containing measured data from the EmStat Pico.

1.5 Communications

In order to use the C library, MSComm, the “extern C” wrapper has to be used because Arduino uses a C++ compiler.

```
extern "C" {
  #include <MSComm.h>
  #include <MathHelpers.C>
};
```

As MSComm is the communication object with the EmStat Pico it needs some read/write functions to be passed in through the MSCommInit(in MSCommInit). However, because the C compiler doesn't understand C++ classes, the write/read functions from the Serial class are wrapped in a normal function, first as shown below.

```
int write_wrapper(char c)
{
  if(_printSent == true)
  {
    //Send all data to PC as well for debugging purposes
    Serial.write(c);
  }
  return Serial1.write(c);
}

int read_wrapper()
{
  int c = Serial1.read();
```

```
  if(_printReceived == true && c != -1) //-1 means no data
  {
    //Send all received data to PC for debugging purposes
    Serial.write(c);
  }
  return c;
}
```

A new UART instance is then created and assigned to TX (14) and RX (13) pins on the Arduino.

```
Uart Serial1(&sercom5, 14, 13, SERCOM_RX_PAD_3, UART_TX_PAD_2);
```

The interrupt handler is attached to the SERCOM.

```
void SERCOM5_Handler()
{
 Serial1.IrqHandler();
}
```

1.5.1 Connecting to the device

The code within the setup() function is executed only once.
Init the Serial and Serial1 ports in here with the baud rate of the EmStat Pico - 230400

```
Serial.begin(230400);
Serial1.begin(230400);
```

Assign the Serial Data Line (SDA) and Serial Clock Line (SCL) to the pins 13 and 14 respectively.

```
//Assign SDA (serial data line) function to pin 13
pinPeripheral(13, PIO_SERCOM_ALT);
//Assign SCL (serial clock line) function to pin 14
pinPeripheral(14, PIO_SERCOM_ALT);
```

The read/write wrapper functions and a new instance of MSComm object are passed on to MSCommInit for initialization.

```
MSComm _msComm;
RetCode code = MSCommInit(&_msComm, &write_wrapper, &read_wrapper);
```

1.5.2 Verifying the connected device

Inorder to verify if the device connected is EmStat Pico, the command to get version string “t\n” is sent to the arduino.

```
const char* CMD_VERSION_STRING = "t\n";
SendScriptToDevice(Cmd_versionString);
void SendScriptToDevice(const char* scriptText)
{
  for(int i = 0; i < strlen(scriptText); i++)
  {
    Serial1.write(scriptText[i]);
  }
}
```

The response from the arduino is is read character by character until the new line character is read to form the version string. If the version string contains the string “esp” it is identified as EmStat Pico.

```
while (Serial1.available())
{
    char incomingByte_pico = Serial1.read();
```

```
    _versionString[i++] = incomingByte_pico;
    if(incomingByte_pico == '\n')
    {
      _versionString[i] = '\0';
      break;
    }
    delay(20);
  }
  if(strstr(_versionString, "espbl")
  {
    Serial.println("EmStat Pico is connected in boot loader mode.");
    return 0;
  }
  else if(strstr(_versionString, "esp"))
  {
    Serial.println("Connected to EmStat Pico.");
    return 1;
  }
  else
  {
    Serial.println("Could not connect to EmStat Pico");
  }
}
```

1.5.3 Sending the method script

The measurement configuration parameters can be either stored in a sd card in the Arduino and read from it or stored in a constant string. For example, in the above example, the measurement configuration parameters are stored in a constant char array as below.

```
//LSV measurement configuration parameters
const char* LSV_METHOD_SCRIPT = "e\n"
                                "var c\n"
                                "var p\n"
                                "set_pgstat_mode 3\n"
                                "set_max_bandwidth 200\n"
                                "set_cr 500u\n"
                                "set_e -500m\n"
                                "cell_on\n"
                                "wait 1\n"
                                "meas_loop_lsv p c -500m 500m 50m 100m\n"
                                "pck_start\n"
                                "pck_add p\n"
                                "pck_add c\n"
                                "pck_end\n"
                                "endloop\n"
                                "cell_off\n\n";
```

The measurement configuration parameters are then sent to the arduino.

```
SendScriptToDevice(LSV_METHOD_SCRIPT);
void SendScriptToDevice(const char* scriptText)
{
  for(int i = 0; i < strlen(scriptText); i++)
  {
    Serial1.write(scriptText[i]);
  }
}
```

1.5.4 Receiving response

The code to receive and parse the response from the device is written in the loop() so that it runs repeatedly and the response can be ontained from the device as and when it is available.

Inorder to read and parse the response from the device, the Receive Package function from the MSComm library can be used in the arduino code as below.

```
while (Serial1.available())
{
    //Read from the device and try to identify and parse a package
    RetCode code = ReceivePackage(&_msComm, &data);
      ...
}
```

The ReceivePackage() function in the C library (MSComm.c) reads character by character until end of line ('\n') from the device using the msComm read wrapper - readCharFunc(). The characters are then combined in to a line of response. For example, in this example, every character read (tempChar) is combined to form a line of response in the response buffer string (buf).

```
int tempChar;                  //Temporary character used for reading
int i = 0;
do {
      tempChar = msComm->readCharFunc();
      if(tempChar > 0)
      {
            buf[i++] = tempChar;          // Store tempchar into buffer
            if(buf[0] == (int)'e')
                  return CODE_RESPONSE_BEGIN;
            if(tempChar == '\n')
            {
                  buf[i] = '\0';
                  if(buf[0] == REPLY_MEASURING)
                        return CODE_MEASURING;
                  else if(strcmp(buf, "*\n") == 0)
                        return CODE_MEASUREMENT_DONE;
                  else if(strcmp(buf, "\n") == 0)
                        return CODE_RESPONSE_END;
                  else if(buf[0] == REPLY_MEASURE_DP)
                        return CODE_OK;
                  else
                        return CODE_NOT_IMPLEMENTED;
            }
      }
} while (i < 99);
buf[i] = '\0';
return CODE_NULL;
```

1.5.5 Parsing the response

Each line of response returned by the function **`ReadBuf`**`() in MSComm.c library`, can be further parsed if it is identified to be a data package. Here's a sample response (raw data) from a Linear sweep voltammetric measurement.

eM0000\n
Pda7F85F3Fu;ba48D503Dp,10,288\n
Pda7F9234Bu;ba4E2C324p,10,288\n
Pda806EC24u;baAE16C6Dp,10,288\n
Pda807B031u;baB360495p,10,288\n
*\n
\n

While parsing the response, various identifiers are used to identify the type of response packages. For example, In the above sample response package,

1. 'e' marks the beginning of a response.
2. 'M' marks the beginning of a measurement loop.
3. 'P' marks the beginning of a row of data package.
4. "*\n" marks the end of measurement.
5. "\n" marks the end of response.

The following information can be found in the data packages received from the device.

- Potential (set cell potential in V)
- Current (measured current in A)

In case of Impedance spectroscopy measurements, the following data values can be obtained from the response.

- Frequency (set frequency in Hz)
- Real part of complex Impedance (measured impedance Ohm)
- Imaginary part of complex Impedance (measured impedance in Ohm)

The following meta data values if present can also be obtained from the data packages.

- CurrentStatus (OK, underload, overload, overload warning)
- CurrentRange (the current range in use at the moment)
- Noise (Noise)

1.5.5.1 Parsing the parameter values

Each row of data package begins with the header 'P'. So the pointer is placed next to 'P' from where the parsing begins.

```
//Identifies the beginning of the response package
char *P = strchr(responsePackageLine, 'P');
char *packageLine = P+1;
```

```
The characters used as delimiters are defined in a char array as below.
```

```
const char delimiters[] = " ;\n";
```

The parameters from the data package line can be then split in to tokens based on the delimiters. In this example, the strtokenize function is used repeatedly to separate the parameters from the package line. It splits the line based on the delimiters, returns the start of the split token and holds the pointer to the next token in the array 'running'. This is being done repeatedly until the function returns null when there are no more tokens or delimiters found or if it is end of the line. The char array 'running' is initialized with the response package line received from ReadBuf().

```
char* running = packageLine;  // Initial index of the line to be tokenized
//Pulls out the parameters separated by the delimiters
char* param = strtokenize(&running, delimiters)
do
{
      //Parses the parameters further to get the meta data values if any
      ParseParam(param, retData);
```

```
}while ((param = strtokenize(&running, delimiters)) != NULL);//Continues
parsing the response line until end of line

char* strtokenize(char** stringp, const char* delim)
{
  char* start = *stringp;
  char* p;
  //Breaks the string when a delimiter is found and returns a pointer with
starting index, Returns NULL if no delimiter is found
  p = (start != NULL) ? strpbrk(start, delim) : NULL;
  if (p == NULL)
  {
    //The pointer to the successive token is set to NULL if no further
tokens or end of string
    *stringp = NULL;
  }
  else
  {
    *p = '\0';
    //Saves the pointer to the beginning of successive token to be further
tokenized
    *stringp = p + 1;

  }
  //Returns the current token found
  return start;
}
```

Each of the parameters separated from the package line can be then parsed to get the actual values of the parameters.

The initial two characters of every parameter identifies the parameter.

```
char paramIdentifier[3];
strncpy(paramIdentifier, param, 2); //Splits the parameter identifier string
paramIdentifier[2] = '\0';
```

For example, in the sample package seen above, the parameter identifiers are
'da7F85F3Fu' - 'da' Potential reading and
'ba48D503Dp,10,288' – 'ba' current reading.

The parameter values hold the next 8 characters.

```
strncpy(paramValue, param+ 2, 8);    //Splits the parameter value string
paramValue[9]= '\0';
```

The parameter value for current reading (8 characters)from the above sample package is '48D503Dp'. This value is further parsed to retrieve the actual parameter value with the respective unit prefix.

```
//Retrieves the actual parameter value
parameterValue = (float)GetParameterValue(paramValue);
```

The SI unit prefix from the package can be obtained from the parameter value at position 8

```
//Identify the SI unit prefix from the package at position 8
char charUnitPrefix = paramValue[7];
```

In the above sample package, the unit prefix for current data is 'p' which is 1e-12 A.

The code below parses the actual parameter value excluding the unit prefix (7 characters) and appends the respective prefixes.

```
char strValue[8];
strncpy(strValue, paramValue, 7);
strValue[7] = '\0';
```

The value is first converted from hex to int

```
char *ptr;
int value = strtol(strValue, &ptr , 16);
```

Then value is then adjusted with the Offset value to receive only positive values.

```
const int OFFSET_VALUE = 0x8000000;
//Values offset to receive only positive values
float parameterValue = value - OFFSET_VALUE;
```

The value of the parameter is returned after appending the SI unit prefix

```
//Return the value of the parameter after appending the SI unit prefix
return (parameterValue * GetUnitPrefixValue(charUnitPrefix));
```

The SI unit prefixes are as follows.

```
const double GetUnitPrefixValue(char charPrefix)
{
	switch(charPrefix)
	{
		case 'a':
			return  1e-18;
		case 'f':
			return 1e-15;
		case 'p':
			return 1e-12;
		case 'n':
			return 1e-9;
		case 'u':
			return 1e-6;
		case 'm':
			return 1e-3;
		case ' ':
			return 1;
		case 'K':
			return 1e3;
		case 'M':
			return 1e6;
		case 'G':
			return 1e9;
		case 'T':
			return 1e12;
		case 'P':
			return 1e15;
		case 'E':
			return 1e18;
	}
	return 0;
}
```

The parameter values are then stored in their respective 'MeasureData' struct (ret_data) member variables based on the identifiers.

The potential readings are identified by the string "da"
The current readings are identified by the string "ba"

```
if(strcmp(paramIdentifier, "da") == 0)
{
      retDdata->potential = parameterValue;
}
else if (strcmp(paramIdentifier, "ba") == 0)
{
      retData->current = parameterValue;
}
```

In case of Impedance sprctroscopy measurement, the following identifiers are used.
The frequency readings are identified by the string "dc"
The real impedance readings are identified by the string "cc"
The imaginary impedance readings are identified by the string "cd"

```
if(strcmp(paramIdentifier, "dc") == 0)
{
      retData->frequency = parameterValue;
}
else if (strcmp(paramIdentifier, "cc") == 0)
{
      retData _data->realImpedance = parameterValue;
}
else if (strcmp(paramIdentifier, "cd") == 0)
{
      ret retData data->imgImpedance = parameterValue;
}
```

After obtaining the parameter identifier and the parameter values from the package, the meta data values if present can be parsed. Meta data values if present are separated by the demiliter character ','

```
//Rest of the parameter is further parsed to get meta data values
ParseMetaDataValues(param + 10, retData);
```

1.5.5.2 Parsing the meta data values

The meta data values are separated based on the delimiter ',' and each of the values is further parsed to get the actual value.

```
const char delimiters[] = ",\n";
char* running = metaDataParams;
char* metaData = strtokenize(&running, delimiters);
```

The first character of each meta data value metaData[0] identifies the type of meta data.

'1' – status
'2' – Current range index
'4' - Noise

The status is 1 character hex bit mask. It is converted to long int. The status can be obtained as shown in the code below.

```
char *ptr;
long statusBits = strtol(&metaDataStatus[1], &ptr , 16);
```

0 indicates OK

```
char* status;
if ((statusBits & 0x0) == STATUS_OK)
      status = "OK";
```

1 indicates overload

```
if ((statusBits & 0x2) == STATUS_OVERLOAD)
      status = "Overload";
```

4 indicates underload

```
if ((statusBits & 0x4) == STATUS_UNDERLOAD)
      status = "Underload";
```

and 8 indicates overload warning (80% of maximum).

```
if ((statusBits & 0x8) == STATUS_OVERLOAD_WARNING)
      status = "Overload warning";
```

For example, in the above sample, the available meta data values for current data are, 10,288. The first meta data value is 10.

1 – meta data status – 0 indicates OK.

The meta data type current range is 2 characters long hex value. If the first bit high (0x80) it indicates a high speed mode current range.

The code below can be used to get current range bits from the package.

```
char  crBytePackage[3];
char* ptr;
//Fetches the current range bits from the package
strncpy(crBytePackage, metaDataCR+1, 2);

The hex value is then converted to int to get the current range string as
shown below.

char* currentRangeStr;
int crByte = strtol(crBytePackage, &ptr, 16);

switch (crByte)
{
      case 0:
            currentRangeStr = "100nA";
            break;
      case 1:
            currentRangeStr = "2uA";
            break;
      case 2:
            currentRangeStr = "4uA";
            break;
      case 3:
            currentRangeStr = "8uA";
            break;
      case 4:
```

```
            currentRangeStr = "16uA";
            break;
      case 5:
            currentRangeStr = "32uA";
            break;
      case 6:
            currentRangeStr = "63uA";
            break;
      case 7:
            currentRangeStr = "125uA";
            break;
      case 8:
            currentRangeStr = "250uA";
            break;
      case 9:
            currentRangeStr = "500uA";
            break;
      case 10:
            currentRangeStr = "1mA";
            break;
      case 11:
            currentRangeStr = "15mA";
            break;
      case 128:
            currentRangeStr = "100nA (High speed)";
            break;
      case 129:
            currentRangeStr = "1uA (High speed)";
            break;
      case 130:
            currentRangeStr = "6uA (High speed)";
            break;
      case 131:
            currentRangeStr = "13uA (High speed)";
            break;
      case 132:
            currentRangeStr = "25uA (High speed)";
            break;
      case 133:
            currentRangeStr = "50uA (High speed)";
            break;
      case 134:
            currentRangeStr = "100uA (High speed)";
            break;
      case 135:
            currentRangeStr = "200uA (High speed)";
            break;
      case 136:
            currentRangeStr = "1mA (High speed)";
            break;
      case 137:
            currentRangeStr = "5mA (High speed)";
            break;
}
```

For example, in the above sample, the second meta data available is 288.

2 – indicates the type – current range
88 – indicates the hex value for current range index – 1mA. The first bit 8 implies that it is high speed mode current range.

1.5.6 Displaying data in the Arduino serial port

The data values and meta data values stored in the 'MeasureData' struct object 'retData' can be then written on the serial port in the arduino code based on the return code from ReceivePackage(). The return code from ReceivePackage() is based on the identified char/string found in the response package.

'e' indicates the beginning of a response and the parsing continues.

```
if(code == CODE_RESPONSE_BEGIN)
{
  //do nothing
}
```

'M' it indicates the beginning of a measurement loop response

```
else if(code == CODE_MEASURING)
{
  Serial.println("\nMeasuring... ");
}
```

'P' indicates the beginning of data package and CODE_OK implies successful parsing of data package.

```
else if(code == CODE_OK)
{
  if(_nDataPoints == 0)
      Serial.println("\nReceiving measurement response:");
  Serial.print("\n");   //Received valid package, print it.
  Serial.print(++_nDataPoints);
  Serial.print("\tE (V): ");
  Serial.print(data.potential, 3);
  Serial.print("\t\ti (A): ");
  Serial.print(sci(data.current, 3));
  Serial.print("\tStatus: ");
  sprintf(readingStatus, "%-15s", data.status);
  Serial.print(readingStatus);
  Serial.print("\tCR: ");
  Serial.print(data.cr);
}
```

"*\n" indicates the end of a measurement loop.

```
else if(code == CODE_MEASUREMENT_DONE)
{
  Serial.println("\n");
  Serial.print("Measurement completed. ");
}
```

'\n' indicates the end of response.

```
else if(code == CODE_RESPONSE_END)
{
  Serial.print(_nDataPoints);
  Serial.print(" data point(s) received.");
}
```

Error message in case parsing fails.

```
else{
      //Failed to parse or identify package.
      Serial.print("\nFailed to parse package: ");
}
```